Janne Vikstedt

Hyväksymistestivetoinen ohjelmistokehitys

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

10.5.2015

Metropolia

Tekijä Otsikko Sivumäärä Aika	Janne Vikstedt Hyväksymistestivetoinen ohjelmistokehitys 30 sivua + 1 liitettä 10.5.2015
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikka
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaaja	projektipäällikkö Kristiina Muhonen yliopettaja Markku Nuutinen

Insinöörityössä oli tavoitteena tutkia, miten hyväksymistestivetoinen ohjelmistokehitys auttaa ohjelmoijan työssä sekä miten Robot Framework -testauskehystä voidaan hyödyntää siinä. Insinöörityön toimeksiantajana oli Eficode Oy, joka on suomalainen ohjelmistoalan yritys.

Insinöörityössä on useita esimerkki sovellustestejä, joilla pyrittiin tutustua sovellustestaamiseen käytännössä. Sovellustesti esimerkkien tekoon käytettiin hyödyksi RSpec-testaustyökalua ja Robot Framework -testauskehystä. Lisäksi käytössä oli Jenkins, joka on jatkuvaa integraatiota varten kehitetty työkalu.

Lopputuloksena saatiin selville, että hyväksymistestivetoinen ohjelmistokehitys oikein käytettynä tarjoaa paljon etuja sekä parantaa projektin onnistumisen mahdollisuutta huomattavasti. Kuitenkin sen käytössä saattaa olla isot riskit, esimerkiksi tilanteessa, jossa projektin kaikki eri osapuolet eivät noudata siinä määritettyjä periaatteita. Lisäksi insinöörityössä huomattiin, että jatkuva integraatio on tärkeä osa hyväksymistestivetoista ohjelmistokehitystä, sillä se vähentää vaadittua työnmäärää ohjelmoijien osalta.

Tutkimuksen ansiosta ymmärretään hyväksymistestivetoisen ohjelmistokehityksen riskit ja osataan varautua niihin riittävällä tavalla. Lisäksi tiedetään hyväksymistestivetoisen ohjelmistokehityksen edut, jolloin osataan ottaa se käyttöön projekteissa, joissa siitä on todellista hyötyä.

Avainsanat	ATDD, hyväksymistestaus, testaus

Author Title Number of Pages Date	Janne Vikstedt Acceptance test-driven development 30 pages + 1 appendix 10 May 2015
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software Engineering
Instructor	Kristiina Muhonen, Project Manager Markku Nuutinen, Principal Lecturer

The goal of this bachelor's thesis was to do research about acceptance test-driven development and see what advantages it brings for software developers and how Robot Framework could be used to write acceptance tests. The idea for this thesis was given by Eficode Oy which is a Finnish software company.

The thesis includes multiple software test examples which were made for the purpose of understanding testing in practice. The examples were made with the RSpec testing framework and Robot Framework. Also Jenkins, a continuous integration tool, was used to execute acceptance tests automatically.

The results showed that the acceptance test-driven development can improve the possibility of projects to succeed a lot. However acceptance test-driven development also has its pitfalls, which can result in project failure if every participant in the project is not following project principles. Also the results showed that continuous integration is an important part of the acceptance test-driven development, since it can be time consuming to execute acceptance tests manually.

Because of this research it will be easier to avoid risks that acceptance test-driven development brings. Also because of knowledge about its advantages, it is easier to understand when it is profitable to use acceptance test-driven development in software development projects.

Keywords	ATDD, acceptance, testing

Sisällys

Lyhenteet

ATDD	Acceptance-test driven development. Ohjelmistokehitystekniikka, jossa pääpainona on hyvä kommunikaatio projektin eri osapuolien välillä.
BDD	Behavior-driven development. Ohjelmistokehitystekniikka, jossa ohjelmistotestit kirjoitetaan selkokielellä.
CI	Continuous integration. Ohjelmistokehitys tapa, jossa sovellustestit suoritetaan automaattisesti muutoksen tapahtuessa.
RF	Robot Framework. Geneerinen ohjelmistotestauskehys, joka on tarkoitettu hyväksymistestivetoiseen ohjelmistokehitykseen ja hyväksymistestien tekoon.
TDD	Test-driven development. Tekniikka jossa kirjoitetaan testit ensin ja sitten vasta itse ohjelmakoodi.

1 Johdanto

Ohjelmistotestaus on yksi ohjelmistokehityksen tärkeimmistä alueista. Ilman kunnollista ohjelmistotestausta ei voida olla varmoja siitä, että sovellus toimii halutulla tavalla. Ohjelmistotestaukseen on käytössä monia eri testausmenetelmiä ja työkaluja. Tämän vuoksi on tärkeää tuntea eri menetelmät ja työkalut, jotta voidaan valita itselle tai omalle projektille ne sopivimmat.

Päätavoitteeni on tutkia, miten hyväksymistestivetoinen ohjelmistokehitys auttaa ohjelmistokehityksessä (Acceptance test-driven development, ATDD). Hyväksymistestivetoisen kehityksen käytöstä saatavat edut vaikuttavat lupaavilta, mutta onko sen ylläpito ja haasteet liian suuri riski projektin onnistumiselle.

Käyn aluksi läpi hieman ohjelmistotestausta yleisesti ja kerron lyhyesti muista yleisistä testaus- ja ohjelmistokehitysmenetelmistä. Pääasiassa tarkoituksena on kuitenkin keskittyä hyväksymistestivetoiseen ohjelmistokehitykseen, kuten tutkia mitkä ovat sen hyödyt ja sudenkuopat. Lisäksi käyn läpi, miten Robot Frameworkia voitaisiin hyödyntää hyväksymistestivetoisessa ohjelmistokehityksessä.

Lopuksi tutkin myös, missä roolissa jatkuva integraatio (Continuous integration, CI) on hyväksymistestivetoisessa kehityksessä sekä miten Jenkins CI-työkalua voitaisiin hyödyntää siinä.

Insinöörityön toimeksiantajana on Eficode Oy. Eficode Oy on vuonna 2005 perustettu suomalaislähtöinen ohjelmistotalo, joka työllistää yhteensä noin 80 ammattilaista Helsingissä, Pekingissä ja Kööpenhaminassa.

2 Ohjelmistotestaus

Ohjelmistotestausta on tehty yhtä kauan kuin ohjelmistokehitystäkin. Ohjelmistotestauksen päätavoite on löytää järjestelmässä olevia virheitä. Kattavalla määrällä sovellustestejä voidaan esimerkiksi todeta, että testattava järjestelmä toimii määritysten määrittämällä tavalla.

Ohjelmistotestauksella voidaan osoittaa esimerkiksi seuraavia asioita [1]: Järjestelmä

- vastaa määritettyjä vaatimuksia
- toimii oikein kaikilla eri syötteillä
- suorittaa toiminnot riittävän nopeasti
- voidaan asentaa ja suorittaa halutussa ympäristössä
- vastaa asiakkaan tarpeita.

Ohjelmistotestausta voidaan tehdä manuaalisesti sekä automaattisesti. Nykyään yritetään automatisoida mahdollisimman suuri osa testeistä, sillä manuaalinen testaus on suuressa osassa tapauksista hitaampaa. Ohjelmistokehityksessä täytyy yleensä samat asiat testata useampaan kertaan, jolloin on järkevää automatisoida testausprosessi. Lisäksi hyvin tehty testi on luotettavampi, sillä ihminen saattaa tehdä manuaalisessa testauksessa huolimattomuusvirheitä.

2.1 Automaattinen ohjelmistotestaus

Automaattisessa ohjelmistotestauksessa tarkoituksena on, että sovellustestaajat tai sovelluskehittäjät kirjoittavat sovellustestejä, jotka testaavat sovelluksen oikeintoiminnallisuutta automaattisesti. Automaattisilla sovellustesteillä saadaan vähennettyä manuaalisen testaamisen tarvetta, sekä mahdollisesti lisättyä testauksen luotettavuutta. Automaattista sovellustestausta varten on kehitetty erilaisia testaustyökaluja, jotka helpottavat automaattisten ohjelmistotestien kirjoittamista sekä tekevät niiden suorittamisesta helpompaa. [2.]

Automaattinen ohjelmistotestaus on nykyään iso osa ohjelmistokehitystä. Sovelluksen automaattitestien kirjoitus kannattaa aloittaa aikaisessa vaiheessa sovelluskehitystä, sillä jälkikäteen sovellustestien kirjoittaminen voi olla vaikeaa, etenkin jos sen joutuu tekemään eri henkilö kuin testattavan ohjelmistokoodin alkuperäinen kehittäjä. [2.]

Täydellisessä maailmassa automaattisia sovellustestejä olisi kirjoitettu jokaiselle sovelluksen toiminnolle. Useimmiten tämä ei kuitenkaan ole mahdollista, sillä yksinkertaisessakin sovelluksessa voi olla lähes loputon määrä mahdollisia eri syötteitä tai tulosteita. Tämän vuoksi ohjelmistotestauksessa on usein käytössä jonkinlainen testaussuunnitelma, jolla valitaan sovellukselle tärkeimmät testattavat kohteet, ottaen kuitenkin huomioon rajallisen määrän aikaa ja resursseja. [2.]

Automaattisten ohjelmistotestien selkeyden ylläpito on tärkeä osa ohjelmistokehitystä. Testien ylläpidosta voi tulla haasteellista, jos testit sisältävät paljon duplikaatti ohjelmakoodia, tai ovat muuten epäselvät. Usein joudutaan tekemään muutoksia ominaisuuksien toimivuuteen, tällöin ohjelmistotestit saattavat mennä rikki. Rikkinäisten ohjelmistotestien korjaaminen voi olla työlästä, jos testejä on vaikea ymmärtää. Epäsiistit ohjelmistotestit saattavat aiheuttaa ohjelmistotestien ylläpidon lopettamisen tai jopa niiden kokonaan hylkäämisen.

2.2 Automaattisen ohjelmistotestauksen hyödyt

Usein automaattisessa ohjelmistotestauksessa testien kirjoittaminen myös auttaa parantamaan sovelluksen toteutusta. Automaattitestejä kirjoittaessa kehittäjä joutuu ajattelemaan sovelluksen toimivuutta eri tavalla, kuin hän ajattelisi ohjelmistokoodia tehdessä ilman testejä. Ohjelmistotestien ansiosta sovelluksen toteutuksesta tulee todennäköisesti parempi, sekä helpommin testattava.

Automaattisia ohjelmistotestejä käytetään usein myös dokumentaationa. Testeistä selviää mitä sovelluksen on tarkoitus tehdä sekä mitä se ei saa tehdä. Tämä on oiva tapa uudelle ohjelmistokehittäjälle projektiin saapuessa tutustua sovelluksen käyttötapauksiin. Pelkkä ohjelmistotestien nimien sekä kuvauksien lukeminen antaa uudelle kehittäjälle hyvän kuvauksen siitä, mitkä ovat sovelluksen pääominaisuudet ja käyttötarkoituk-

set. Lisäksi tutkimalla ohjelmistotestejä tarkemmin voidaan selvittää miten sovelluksen eri ominaisuudet on tehty.

Vaikka testien kirjoitus vie paljon aikaa, se kuitenkin säästää melkein aina aikaa pitkällä aikavälillä. Ohjelmistokehityksessä joudutaan testamaan samat asiat useaan kertaan, sillä muutokset sovellukseen vaikuttavat useaan paikkaan ja sovelluksen oikein toiminnallisuus täytyy todentaa uudelleen. Tämän vuoksi automaattinen ohjelmistotestaus on nopeaa, koska testit voidaan suorittaa yhdellä komennolla.

Lisäksi automaattiset ohjelmistotestit antavat ohjelmistokehittäjälle varmuutta sovelluksen kunnosta. Isoissa sovelluksissa pieni muutos saattaa rikkoa sovelluksen ennalta arvaamattomassa paikassa. Varsinkin uudella ohjelmistokehittäjällä, joka ei täysin tunne sovellusta, voi olla epävarmuutta muutoksia tehdessä. Kattavalla määrällä automaattitestejä voidaan parantaa itseluottamusta ja todeta, että sovellus toimii halutulla tavalla vielä muutoksen jälkeenkin.

2.3 Yksikkötestiesimerkki

```
1  describe Calculator do
2    let(:calculator){Calculator.new}
3    context "sum()" do
4      it "sums two numbers together" do
5        result = calculator.add(1, 2)
6        expect(result).to eq(3)
7      end
8    end
9    context "multiply()" do
10     it "multiplies two numbers together" do
11       result = calculator.multiply(2, 2)
12       expect(result).to eq(4)
13     end
14   end
15 end
```

Esimerkki 1. Laskimen yksikkötesti esimerkki RSpec -työkalulla.

Esimerkissä 1 on esitetty yksinkertainen esimerkki yksikkötestistä. Yksikkötestissä testataan luokan Calculator kahta eri metodia sum ja multiply. Avainsanat describe ja context luovat testiryhmän, jolla ei sinänsä ole vaikutusta itse testeihin, mutta testeistä ja testiraporteista tulee helpommin luettavia. Itse testit tapahtuvat rivillä 6 ja 12, jossa tehdään oletus, että muuttuja result vastaa haluttua arvoa.

Yleisimmissä testityökaluissa on käytössä punainen ja vihreä väri. Punainen väri tarkoittaa epäonnistunutta testiä ja vihreä onnistunutta.

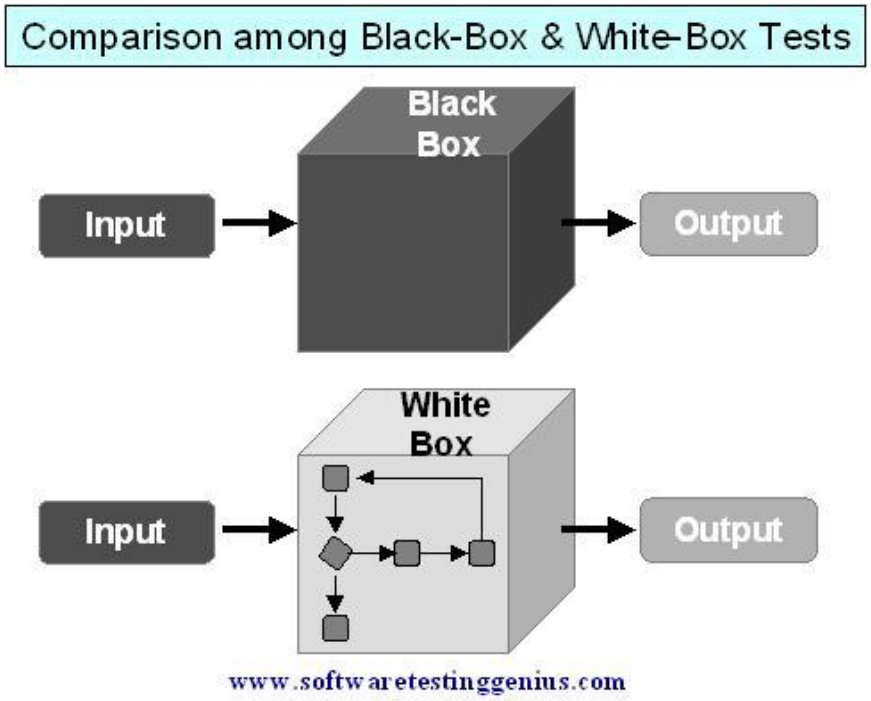
```
Calculator
  sum()
    sums two numbers together
  multiply()
    multiplies two numbers together (FAILED - 1)

Failures:
  1) Calculator multiply() multiplies two numbers together
       expected: 4
            got: 3

Finished in 0.02229 seconds (files took 1.6 seconds to load)
2 examples, 1 failure
```

Esimerkki 2. RSpec työkalun tuottama testiraportti.

Esimerkissä 2 on testiraportti esimerkin 1 ajetuista yksikkötesteistä. Testiraportista voidaan todeta, että testejä on yhteensä kaksi ja niistä toinen on epäonnistunut. Värien avulla nähdään selvästi, missä ongelma on, sillä se on merkattu punaisella värillä. Tässä tapauksessa virheen syy on, että metodi multiply on palauttanut arvon 3, vaikka oletettiin arvon olevan 4.

3 Ohjelmistotestauksen tasot

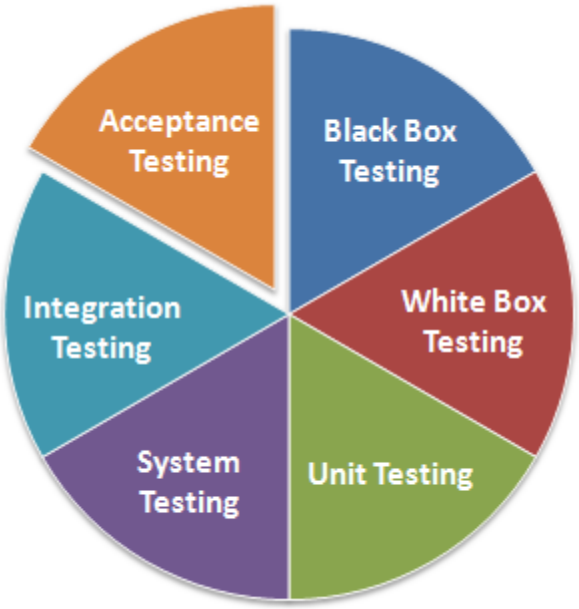
Ohjelmistotestauksessa testaustasot jaetaan pääsääntöisesti kahteen kategoriaan: sisäinen testaus (white-box) sekä ulkoinen testaus (black-box).



Kuva 1. Sisäinen ja ulkoinen testaus [16.]

Ulkoisella testauksella tarkoitetaan testausta, jossa sovellusta testataan ylimmällä tasolla, eikä siinä olla tietoisia sovelluksen sisäisestä toteutuksesta. Tämä tapahtuu usein sovelluksen käyttöliittymätasolla, jossa sovellusta käytetään käyttäjän tavoin. Ulkoisessa testauksessa löydetyt ongelmat eivät aina paljasta virheiden syytä, vaan virheiden syyn löytämiseen voidaan joutua käyttämään sisäistä testausta.

Sisäisessä testauksessa ollaan tietoisia sovelluksen sisäisestä toteutuksesta, kuten kuvasta 1 voidaan todeta. Sisäinen testaus ei välttämättä paljasta virheitä suuremmalta kokonaisuudelta, sillä siinä pyritään usein testaamaan sovellusta pienemmissä osissa. [3]



Kuva 2. Testaustasot [4].

Kuvassa 2 on esitetty eri testitasot. Sisäisen ja ulkoisen testauksen alle lajitellaan pääsääntöisesti neljä testaus tasoa: yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksymistestaus. Näistä osa voidaan lajitella molempiin, sisäiseen ja ulkoiseen testaukseen.

3.1 Yksikkötestaus

Yksikkötestit (Unit tests) ovat testejä, jotka testaavat sovelluksen pieniä osia. Olio-ohjelmoinnissa yksikkötesti voi testata kokonaista luokkaa, tai yhtä metodia. Tämä ei kuitenkaan tarkoita sitä, että yhdellä metodilla ei voisi olla montaa testiä. Usein yksikkötesteissä kirjoitetaankin monta testiä yhtä metodia varten, jolloin voidaan testata testattava kohde monilla eri syötteillä. Yksikkötestaus pääsääntöisesti lajitellaan sisäisen testauksen alle. [4.]

Yksikkötestit testaavat pääsääntöisesti, että sovelluksen komponentit toimivat yksinään. Tällöin virheiden löytäminen on helpompaa, sillä näin voidaan testata sovellusta pienimmissä osissa. [5.]

Yleensä yksikkötestit kirjoittaa ohjelmoija. Ohjelmoija voi todeta yksikkötestin avulla, että tehty toiminnallisuus toimii oikein kaikilla mahdollisilla eri syötteillä. Tällöin ohjelmoija saa lisää varmuutta siitä, että toteutus pystyy tarjoamaan halutun toiminnallisuuden, ja se toimii kaikilla eri syötteillä.

3.2 Integraatiotestaus

Integraatiotestaus (Integration testing) tapahtuu yleensä yksikkötestauksen jälkeen. Sen tarkoituksena on yhdistää sovelluksen yksittäiset moduulit ja testata niitä kokonaisuutena, eli varmistaa että ne toimivat oikein myös yhdessä.

Integraatiotestauksessa voidaan käyttää kahta testausmenetelmää: Big Bang tai nouseva testaus. Big Bang -menetelmässä kaikki sovelluksen moduulit testataan kerralla, eli testit kirjoitetaan vasta kun kaikki ovat valmiina.

Nousevassa testauksessa moduulit testataan heti moduulin valmistuessa. Moduulit jotka eivät ole vielä valmiita, voidaan korvata tyngällä (stub). Tynkä on testauksessa käytettävä valemoduuli. Tyngän tehtävä on toteuttaa kohdemoduulin rajapinta, mutta palauttaa arvoja ennalta määritellyllä tavalla. Tyngän ansiosta voidaan testata välittämättä moduulin riippuvuuksista sekä testata moduulin toimivuutta, vaikka riippuvuudet olisivat vielä keskeneräisiä. Moduulin valmistuessa tynkä voidaan korvata oikealla moduulilla. [19.]

3.3 Järjestelmätestaus

Järjestelmätestien (System testing) tavoitteena on tarkastella sovelluksen toimivuutta kokonaisuutena. Testitapaukset suunnitellaan riskien ja/tai vaatimusten perusteella. Järjestelmätestausta voidaan pitää viimeisenä vaiheena jonka kehitystiimi suorittaa sovellukselle. Yleensä järjestelmätestaukseen siirrytään vasta kun sovellus on valmis ja integraatiotestaus on suoritettu. [5.]

3.4 Hyväksymistestaus

Hyväksymistestauksessa (Acceptance testing) on paljon samaa kuin järjestelmätestauksessa. Sen tavoite on todeta, että järjestelmä on valmis ja se toteuttaa määritetyt vaatimukset. Hyväksymistestit suoritetaan usein järjestelmätestauksen jälkeen ja sen suorittaa asiakas, tai sovelluksen käyttäjät. [6.]

Hyväksymistestit määritellään käyttötapauksien perusteella. Ketterässä ohjelmistokehityksessä (kts. sivu 10) tämä tarkoittaisi sitä, että iteraatioon valituille käyttäjätarinoille tehdään hyväksymistestit. Hyväksymistestien määrää ei ole rajattu, vaan niitä suoritetaan tarvittava määrä, jotta voidaan todeta sovelluksen oikein toimivuus. [6.]

Hyväksymistestit kuuluvat ulkoiseen testaukseen, eli niissä ei oteta kantaa sovelluksen toteutukseen tai sisäiseen toimintaan, vaan se testaa sovellusta ylimmällä tasolla, eli esimerkiksi käyttöliittymätasolla. [6.]

Hyväksymistesteissä voidaan vahvistaa esimerkiksi seuraavia asioita:

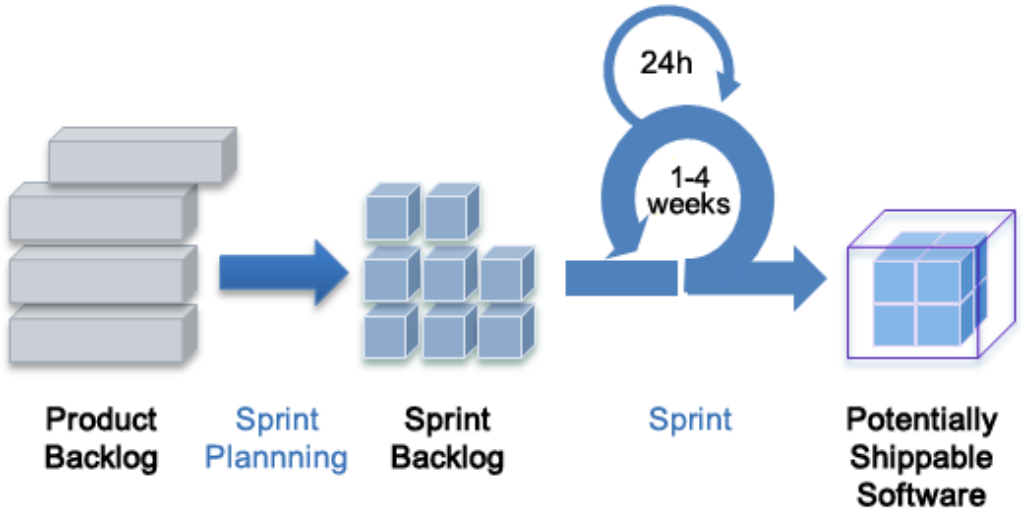
Saatavuus ja luotettavuus. Onko tuote aina saatavilla ja kestääkö se tarvittavan kuorman. Tätä voidaan testata esimerkiksi stressitesteillä.

Käytettävyys. Onko tuotteen käyttöliittymä määrityksien mukainen ja toimiiko se käytännössä oletetulla tavalla, sekä onko sovelluksen visuaalinen ulkoasu riittävä.

Tietoturva. Eri sovelluksilla on erilaiset tietoturva vaatimukset. Tietoturva testataan kyseisen sovelluksen tietoturva vaatimusten mukaisesti.

Ylläpito. Yleensä sovellukset vaativat jonkunlaista ylläpitoa julkaisun jälkeen. Hyväksymistesteissä voidaan varmistaa, että sovelluksen päivitys pystytään hoitaa onnistuneesti sekä riittävän nopeasti.

4 Ketterä ohjelmistokehitys



Kuva 3. Scrum -prosessi. [15]

Viime vuosien kuuma trendi on agile eli ketterät ohjelmistokehitys menetelmät. Useat ketterät menetelmät jakavat perinteisen vesiputousmallin prosessin useaksi lyhyeksi iteraatioksi. Yksi iteraatio kestää yleensä yhdestä neljään viikkoon ja tavoitteena on saada jokaisen iteraation jälkeen tuote, joka on julkaisukelpoinen. Yleisesti käytettyihin ketteriin menetelmiin kuuluu esimerkiksi Extreme Programming (XP) ja Scrum. Kuvassa 3 on esimerkki Scrum projektin eri vaiheista.

Agile Manifesto toi esille ensimmäisenä ketterän ohjelmistokehityksen periaatteet. Manifest sai alkunsa vuonna 2001, kun 17 ohjelmistokehittäjää julkaisivat listan 12 keskeisestä periaatteesta. Kaikki agilen toteuttavat ohjelmistokehitysmenetelmät seuraavat neljää tyypillistä arvoa [7];

- *Yksilöitä ja kanssakäymistä* enemmän kuin menetelmiä ja työkaluja
- *Toimivaa ohjelmistoa* enemmän kuin kattavaa dokumentaatiota
- *Asiakasyhteistyötä* enemmän kuin sopimusneuvotteluja
- *Vastaamista muutokseen* enemmän kuin pitäytymistä suunnitelmassa

Ketterissä menetelmissä jokainen iteraatio sisältää kaikki perinteiset ohjelmistoprojektin tarvittavat tehtävät: suunnittelu, ohjelmointi, testaus ja dokumentointi. Ketterät menetelmät pyrkivät hyvään automaatiotestaukseen, nopeaan muutokseen reagointiin, kommunikaatioon ja hyvään tiimityöhön.

Ketterässä ohjelmistokehityksessä pyritään käyttämään resurssit sinne missä siitä on eniten hyötyä, jolloin sovellusprojekti pystyy minimoimaan riskien syntymistä.

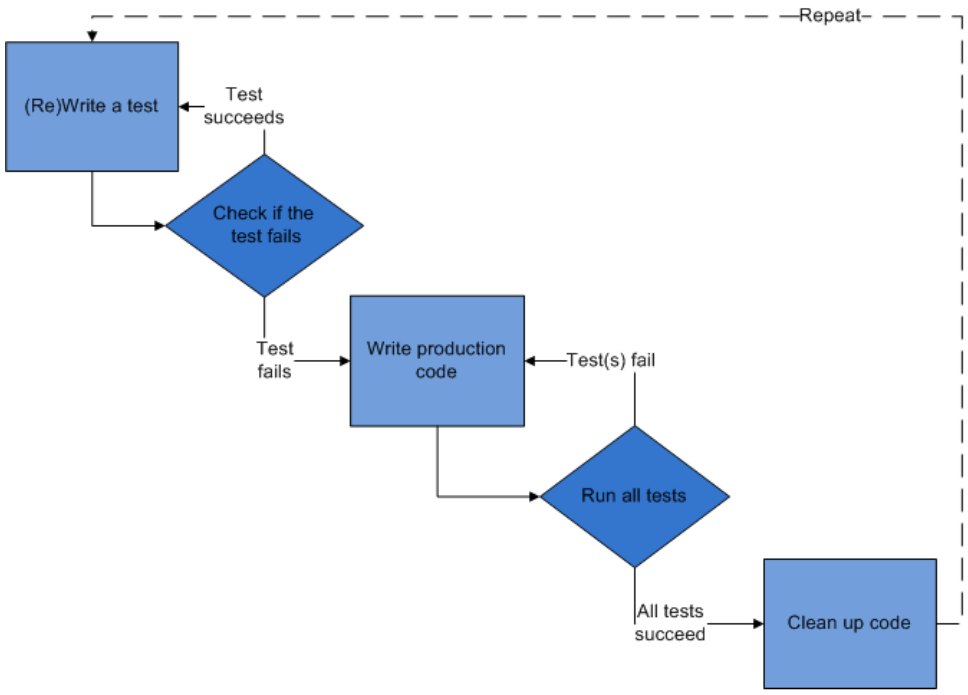
5 Testivetoinen ja käyttytymislähtöinen kehitys

5.1 Testivetoinen ohjelmistokehitys

Testivetoinen ohjelmistokehitys esiteltiin ensimmäisen kerran erillään Extreme Programmingista vuonna 2003 Kent Beckin kirjassa ”Test-Driven Development by Example” [9].

Perinteisessä ohjelmistokehityksessä kirjoitetaan ohjelmakoodi ensin ja sitten vasta testit. Kun sovellus tehdään tähän tyyliin, saattaa ilmetä vaikeuksia testien kirjoituksessa, sillä testaamista ei ole luultavasti ole mietitty tarkkaan toteutusta tehdessä. Ongelmana tässä on myös tilanne, jossa testit on kirjoitettu vastaamaan ohjelmakoodia, mutta ei itse sovelluksen määrityksiä.

Testivetoisessa ohjelmistokehityksessä kirjoitetaan aina testit ensin. Tällöin sovelluksen kulkua ohjaavat testit, ja ohjelmakoodin toteutuksesta tulee todennäköisesti parempi. Lisäksi testien kirjoittaminen ensin auttaa ehkäisemään virheitä syntymästä heti kehityksen alussa. Mitä pidemmälle ohjelma etenee ennen virheiden korjaamista, sitä suuremman työn se vaatii. Täydellisessä testivetoisessa kehityksessä ei uutta ohjelmakoodia sovellukseen lisätä, ellei ensin ole olemassa testi, joka antaa virheen. [9.]



Kuva 4. Testivetoisen ohjelmistokehityksen vaiheet [17].

Testivetoinen ohjelmistokehitys sisältää kolme vaihetta: testin kirjoitus, ohjelmakoodin kirjoitus ja refaktorointi.

Testivetoisessa ohjelmistokehityksessä uuden ominaisuuden kehittäminen alkaa aina testin kirjoituksella. Testit kirjoitetaan kohde ominaisuuden määrityksien perusteella. Tällöin ohjelmakoodin kirjoitusta voidaan ohjata testien avulla, jolloin toteutuksesta tulee helpommin testattava sekä todennäköisesti parempi rakenteeltaan.

Kun on olemassa testi, joka antaa virheen, voidaan alkaa kirjoittamaan ohjelmakoodia. Ohjelmakoodiin tehdään minimaalinen lisäys tai muutos, jotta testit saadaan vihreäksi, eli ajettua onnistuneesti. Ohjelmakoodin siisteyteen ei painoteta liikaa aikaa tässä vaiheessa. Tämä johtuu siitä, että refaktorointi vie ylimääräistä aikaa, ja se on hyvä suorittaa vasta, kun isompi kokonaisuus on valmis, jolloin ymmärretään kokonaisuus paremmin.

Kun tietty kokonaisuus on valmis, kuten luokka, sekä kaikki testit menevät läpi onnistuneesti, ollaan valmiita suorittamaan viimeinen vaihe eli refaktorointi. Refaktoroinnilla tarkoitetaan koodin siivoamista, ilman että toiminnallisuutta muutetaan. Tämä voi siis tarkoittaa esimerkiksi duplikaatti koodin poistamista tai suurten metodien pilkkomista

pienempiin osiin. Refaktorointi suoritetaan vasta tietyn kokonaisuuden lopuksi, sillä jokaisen testin jälkeen suoritettu refaktorointi saattaa aiheuttaa turhaa työtä tai monimutkaista ohjelmakoodia. Ohjelmoijalla on parempi kuva kokonaisuudesta, kun ominaisuus on hyvin testattu, sekä ohjelmakoodi toimii oikein.

Aloittelevilla testivetoisen ohjelmistokehityksen käyttäjillä ilmenee usein ongelmia ymmärtää miten voi kirjoittaa testin, jos ominaisuutta, eikä ohjelmakoodia ole vielä tehty. Testeillä tulisi määrittää miten ominaisuuden tulisi toimia kaikissa eri tilanteissa, eli tämä vaatii asian miettimistä tarkemmin etukäteen. [20.]

5.2 Käyttäytymislähtöinen kehitys

Käyttäytymislähtöisessä kehityksessä (Behavior-driven development, BDD) on hyvin paljon samaa kuin TDD:ssä. BDD pyrkii lisäämään kommunikaatiota asiakkaan ja kehittäjien välille. BDD:ssä testit luodaan kaikkien ymmärrettävällä kielellä, eli arkikielellä (engl .ubiquitous language). Tällöin asiakkaat pääsevät paremmin mukaan sovelluskehitykseen. [9.]

Kuten TDD, myös BDD sopii hyvin ketteriin ohjelmistoprojekteihin. BDD:ssä asiakas ja sovelluskehittäjät yhdessä määrittävät selkokieliset testit käyttötapauksia vastaaviksi. Myöhemmin sovelluskehittäjät rakentavat järjestelmän, joka tekee selkokielisistä testeistä toimivia testejä. [10.]

6 Hyväksymistestivetoinen kehitys

Tässä luvussa käyn läpi, mitä hyväksymistestivetoinen kehitys (Acceptance test-driven development, ATDD) on, mitä hyötyä sen käytöstä on ja mitkä ovat yleisimmät haasteet sen käytössä.

ATDD yhdistää sovelluksen käyttäjätarinat ja automaattisen testauksen. Tarkoituksena on, että kehityksen alkuvaiheessa kehittäjät ja asiakas määrittelevät sovelluksen vaatimukset. Vaatimuksia voidaan hyödyntää myöhemmin sovelluksen kehitys vaiheessa.

Hyväksymistestivetoisessa kehityksessä asiakas on tiiviisti mukana hyväksymistestien suunnittelussa. Usein on myös toivottavaa, että asiakas osallistuu automaattisten hyväksymistestien tekoon. Tällä tavalla saadaan varmistettua, että testit ovat oikein tehtyjä, jolloin niihin voidaan luottaa enemmän. Käytännössä tämä ei kuitenkaan ole helppoa, sillä asiakas ei usein halua tehdä testejä, koska se vaatii aikaa ja teknistä osaamista. Tämän vuoksi ATDD -työkalut on yleensä suunniteltu helppokäyttöisiksi, jotta myös vähän teknistä taitoa omaavat henkilöt voisivat niitä käyttää. [9.]

6.1 Hyödyt

Hyväksymistestivetoisen kehityksellä on seuraavat kolme päätavoitetta: kommunikaatio, dokumentaatio ja se, että sovellus pysyy vaaditussa tilassa.

Luultavasti suurin hyöty tulee esille kommunikaatiomuodossa. ATDD pakottaa projektin eri osapuolia kommunikoimaan keskenään. Jokainen uusi käyttäjätarina täytyy määritellä tarkoin, jotta siitä voidaan luoda tarkat hyväksymistestit. Hyvä kommunikaatio sovelluskehitysprojekteissa on suuri etu. Kun sovelluskehittäjät ymmärtävät hyvin asiakkaan tarpeen ja sovelluksen käyttötarkoituksen, väärinymmärrykset pienenevät ja kehittäjät voivat paremmin auttaa asiakasta. [11.]

Yleinen ongelma ohjelmistokehitys projekteissa on vanhentunut dokumentaatio. Kehitysvaiheessa oleva sovellus muuttuu paljon, siksi dokumentaation ylläpitäminen vie paljon aikaa. Ohjelmistokehittäjät eivät yleensä pidä dokumentaatioiden kirjoittelusta, vaan koodin kirjoittamisesta. [11.]

Oikein noudatetussa ATDD:ssä sovelluksen dokumentaatio pysyy aina ajan tasalla, sillä hyväksymistestit toimivan myös dokumentaationa. Hyväksymistesteistä ilmenee heti mitä sovelluksella voidaan tehdä. Hyväksymistestit eivät vanhene, niin kauan kuin niitä käytetään. [11.]

Viimeisenä ATDD pitää huolen siitä, että sovellus pysyy vaatimusmäärittelyiden vaatimassa tilassa koko kehityksen ajan, sekä ylläpitovaiheessa. Läpi menevät hyväksymistestit antavat ohjelmistokehittäjälle varmuutta uuden koodin oikein toiminnallisuudesta, sekä varmistuksen siitä, että sovellus toimii edelleen muutoksen jälkeenkin. [11.]

6.2 Yleisimmät haasteet ATDD:ssä

ATDD lupaa sekä myös palkitsee paljon, kun sitä noudatetaan oikein. Mutta täydellisen ATDD:een oikein noudattaminen saattaa koitua haasteelliseksi. Yleinen ongelma on, että kiireessä kehittäjältä testien kirjoitus jää tekemättä ja myöhemmin niitä on ikävä tehdä, tai ei ole riittävästi aikaa. Kuten TDD:ssä, myös ATTD:ssä on tärkeää, että hyväksymistestit ohjaavat kehityksen kulkua, eikä testejä kirjoiteta koodin perusteella. [11.]

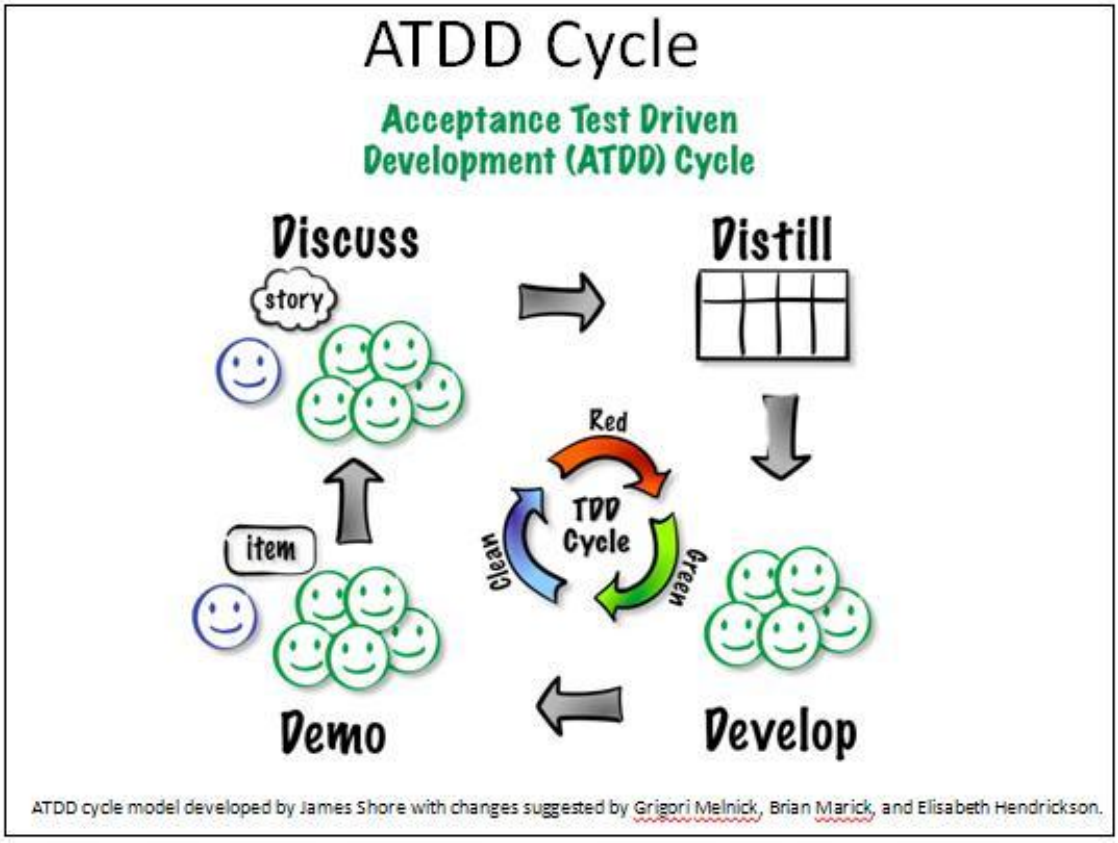
Koska ATDD vie paljon aikaa, se vaatii kaikkien järjestelmän kehitykseen osallistuvien vankkaa sitoutumista. Työtä on liian paljon, että testejä voitaisiin ylläpitää ilman että kaikki ovat siihen osallisena. [11.]

Hyväksymistestien suorittaminen voi olla todella hidasta, varsinkin kun sovelluksen koko kasvaa. Tämä on vaarallista, sillä hitaiden testien suorittamisesta ei kukaan pidä ja testit saattavat jäädä suorittamatta. Tämän vuoksi on tärkeää, että taustalla on jonkinlainen jatkuva integraatio (CI) –järjestelmä, joka suorittaa testejä automaattisesti ja ilmoittaa virheellisistä testeistä. [11.]

Hyväksymistestien ylläpito saattaa ilmetä myös haasteelliseksi. Hyväksymistestit pyritään pitämään siisteinä, sekä duplikaatti koodia pyritään välttämään. Huonokuntoisia testejä on ikävä ylläpitää ja se saattaakin johtaa testien hylkäämiseen. Kun sovellus kasvaa, kasvaa myös hyväksymistestit. [11.]

6.3 ATDD:n vaiheet

Tässä luvussa käyn läpi ATDD:n vaiheet sekä näytän esimerkkejä, miten asia voitaisiin hoitaa käyttäen Robot Framework -testaus työkalua. Esimerkeissäni hyödynnän yksinkertaista verkkokauppa www sivua, jonka olen toteuttanut tätä varten. Verkkosivujen testaus ei yksinään RF:n avulla onnistu, vaan tarvitaan avuksi kirjasto Selenium 2. Selenium 2 mahdollistaa verkkosivujen käyttämisen ohjelmakoodin avulla. Se tarkoittaa, että voidaan esimerkiksi mennä verkkosivuille, täyttää tekstikenttiä tai lukea verkkosivun sisältöä.



Kuva 5. Hyväksymistestivetoisen ohjelmistokehitys projektin kulku [18].

Suunnittelu ja keskustelu. Tämä on ATDD -prosessin ensimmäinen vaihe, mutta se saattaa olla jopa tärkein. Tässä vaiheessa selvitetään halutun tuotteen vaatimukset. Mukana ovat tuotteen omistaja, tai joku korvaava henkilö. Lisäksi mukana on mikä tahansa sidosryhmä, jolla on potentiaalista tietoa sovelluksen vaatimuksista ja itse sovelluskehitys ryhmä.

Vaatimukset määritellään käyttäjän näkökulmasta, eli käyttöliittymä tasolla. Tärkeää on, että sovelluskehitystiimi saa hyvän ymmärryksen siitä, mitä sovelluksella tavoitellaan. Yleinen kysymys, joka voidaan esittää tässä tilanteessa on ”Kuvittele, että järjestelmä on valmis. Kuinka käyttäisit sitä ja mitä olettaisit siltä?”. Tämän tyyppisillä kysymyksillä saadaan aikaiseksi esimerkkejä sovelluksen käyttötapauksista, jotka voidaan muuntaa hyväksymistesteiksi. Käyttötapauksia voidaan kutsua myös käyttäjätarinoiksi esimerkiksi ketterässä ohjelmistokehityksessä. [12.]

Esimerkkejä käyttötapauksista verkkokauppa sovelluksessa:

- Käyttäjä voi selata tuotteita.
- Käyttäjä voi lisätä tuotteita ostoskoriin.
- Käyttäjä voi tilata tuotteen / tuotteita.

Käyttötapaukset testeiksi. Seuraavaksi käyttötapaukset muutetaan testeiksi, jotka voidaan suorittaa. Tarkoitus ei kuitenkaan ole vielä toteuttaa itse testikoodia, joka kommunikoi sovelluksen kanssa. Tarkoituksena on kirjoittaa ylimmäntason esimerkit, sekä pystyttää testaus ympäristö, jotta testit voidaan ajaa ja nähdä kun ne epäonnistuvat. Tässä vaiheessa pyritään pitämään asiakas mukana, jotta voidaan varmistua siitä, että testit on toteutettu oikein.

browse_products.robot

```
1   *** Settings ***
2   Resource          resources/${ENVIRONMENT}.robot
3   Suite Setup       Open Browser And Open Frontpage
4   Suite Teardown    Close Browser
5
6   *** Test Cases ***
7   Browse Categories
8     Click 'Cameras' Category
9     Confirm 'Cameras' Page Subtitle
10    Click 'Games' Category
11    Confirm 'Games' Page Subtitle
```

Esimerkki 3. Robot Framework käyttötapaus testiksi.

Esimerkissä 3 on Robot Frameworkilla tehty esimerkki siitä, miten käyttötapaus ”Käyttäjä voi selata tuotteita” voitaisiin toteuttaa. Esimerkissä on käytetty vain selväkielisiä testin nimiä ja avainsanoja. Tällä tavalla on helppo ymmärtää mitä testit tekevät, vaikka lukija ei tietäisi RF:n käytöstä mitään. Tässä vaiheessa ei tarvita vielä liimakoodia, joka sitoo testit ja itse sovelluksen, vaan se tapahtuu vasta seuraavassa vaiheessa ohjelmakoodin kirjoituksen rinnalla.

Ohjelmakoodin kirjoitus. Tavoitteena on käyttää ”Testit ensin” -lähestymistapaa. Kehittäjä ajaa testit ensin ja näkee, missä tulee ensimmäinen virhe ja sitten korjaa sen. Itse ohjelmakoodin kehityksen rinnalla tehdään myös hyväksymistestien liimakoodi, joka kommunikoi sovelluksen kanssa käyttöliittymätasolla.

Tämän lisäksi voidaan tehdä alemman tason testejä, kuten yksikkötestejä. Myös dokumentaatiota voidaan päivittää tässä vaiheessa kyseisessä projektissa sovituin ehdoin.

resources/common_keywords.robot

```
1   *** Keywords ***
2   Click '${category}' Category
3     Click Link  css=#categories a.${category}
4
5   Confirm Subtitle  [Arguments]  ${text}
6     ${subtitle} =  Get Text  id=subtitle
7     Should End With  ${text}  ${subtitle}
```

Esimerkki 4. Robot Framework yhteiset avainsanat

Esimerkissä 4 on toteutettu liimakoodi ”tuotteiden valinta” -hyväksymistestille. Esimerkissä on luotu uusi avainsana, joka valitsee halutun tuotteen. Lisätään tämän avainsanan yhteisiin avainsanoihin, sillä vastaavaa avainsanaa tullaan todennäköisesti tarvitsemaan muissakin hyväksymistesteissä. Käytetään tässä avuksi Selenium 2 -kirjaston avainsanaa ”Click Link”. Etsitään haluttu linkki verkkosivulta css -määrityksien avulla ja painetaan sitä.

Lisäksi esimerkissä teemme avainsanan ”Confirm Subtitle”. Avainsana ottaa argumentin ”text” ja vertailee sitä sivulla olevaan html -elementin ”subtitle” -arvoon.

browse_products.robot

```
1  *** Keywords ***
2  Confirm '${category}' Page Subtitle
3    Wait Until Keyword Succeeds  5 s  .5 s  Confirm Subtitle  ${category}
```

Esimerkki 5. Robot Framework suiten sisäinen avainsana

Esimerkissä 5 luodaan itse kyseisen testin kanssa samaan tiedostoon avainsanan. Tämä avainsana sijoitetaan testin kanssa samaan tiedostoon, sillä kyseistä avainsanaa ei toistaiseksi tarvita muualla. Tämä avainsana pitää huolen, että testi ei epäonnistu siksi, että sivu ei ollut vielä kerinnyt latautua. Kutsutaan avainsanaa ”Confirm Subtitle” puolen sekunnin välein. Testi epäonnistuu, jos aikaa kuluu yli viisi sekuntia.

Tuotteen esittely. Kun testit menevät läpi, voidaan sovellus esittää sovelluksen omistajalle ja muille sidosryhmille. Tämä saattaa tuottaa uusia vaatimuksia tai muuttaa vanho-

ja. Testien avulla voidaan todeta, että sovellus on tehty määrityksien määrittämällä tavalla. [12.]

7 Robot Framework

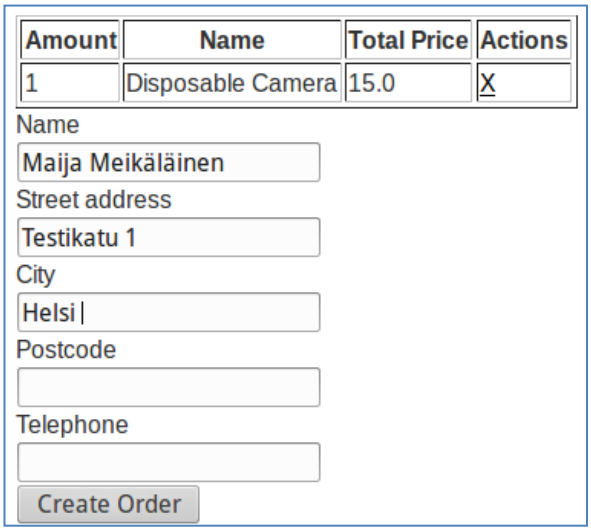
Robot Framework on automaatiotestaustyökalu hyväksymistestausta ja hyväksymistestivetoista kehitystä varten. Se on avoimen lähdekoodin ohjelmistotestaus kehys, ja se on julkaistu Apacha License 2.0 -lisenssillä. RF on käyttöjärjestelmästä ja sovelluksesta riippumaton, ja se on kehitetty Python -ohjelmointikielellä.

RF hyödyntää helposti käytettävää taulukkomaista testaus syntaksia ja avainsana vetoista lähestymistapaa. RF:n päätavoitteet on hyväksymistestien helppolukuisuus, helppo ylläpidettävyys ja nopeasti suoritettavat hyväksymistestit. [13.]

RF ei vaadi käyttäjältä paljoa teknistä osaamista, ja siksi testejä voi luoda esimerkiksi tuotteen omistaja, jolla on eniten tietoa sovelluksen käyttötarkoituksesta.

RF:ssä on hyvä laajennustuki, ja siksi se mahdollistaa kaiken tyyppisten sovellusten testaamisen. Laajennuksia voidaan kirjoittaa itse tai käyttää jo olemassa olevia. Yksi yleisesti käytetty kirjasto on Selenium 2, joka tekee verkkosovellus testauksesta mahdollista.

7.1 Hyväksymistestit käytännössä



Kuva 6. Robot Framework ja Selenium 2 hyväksymistesti esimerkki.

Hyväksymistestit suoritetaan käyttöliittymä tasolla. Verkkosovelluksen hyväksymistesteissä tämä tarkoittaa sitä, että testit avaavat verkkoselaimen ja alkavat käyttää sovellusta kuten normaali käyttäjä.

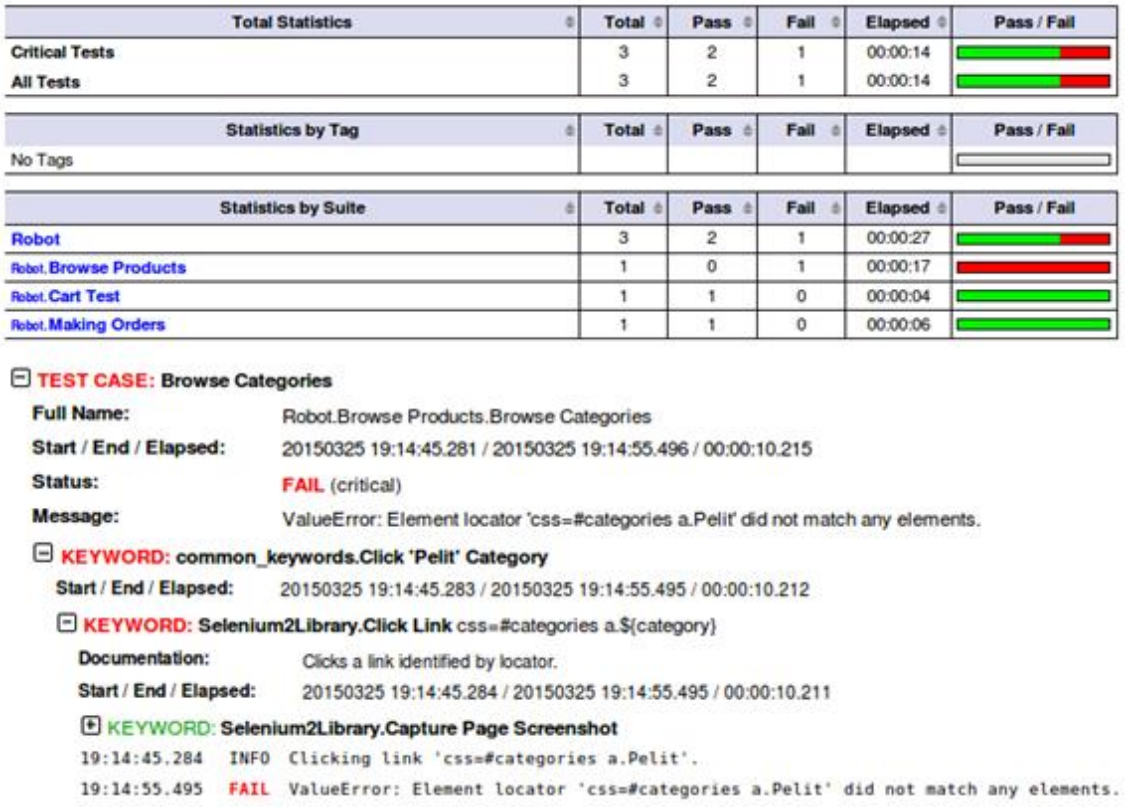
Kuvassa 6 on käynnissä oleva hyväksymistesti, jossa testit täyttävät lomaketta tuotteen tilausta varten. Käytännössä tämä tapahtuu todella nopeasti, eikä testaukseen mene paljoa aikaa.

7.2 Tagit

Tagit ovat mukava osa Robot Frameworkiä. Tagien avulla voidaan ohjata, mitä testejä suoritetaan. Yleensä automaattitestauksessa halutaan suorittaa kaikki testit, mutta kuitenkin joskus voi olla tilanteita, joissa on tarve suorittaa hyväksymistestit esimerkiksi tuotantoversiota vasten. Tällöin ongelmana voi olla, että joitakin testejä ei haluta suorittaa, esimerkiksi ei haluta poistaa tai muokata sovelluksen tärkeitä tietoja. RF:n testeille voidaan antaa tageja, jotka ohjaavat, mitkä testit suoritetaan. Esimerkiksi voitaisiin antaa tagi ”Production” kaikille testeille, jotka voidaan suorittaa tuotantoympäristössä.

7.3 Testiraportit

Robot Framework luo testeistä selkeät testiraportit. Tiedostoista näkee tarkasti testit jotka onnistuivat ja mitkä epäonnistuivat, sekä syyt epäonnistumiselle. Kuvassa 7 on ilmennyt yksi virhe testissä ”Browse Categories”. Testi yrittää etsiä verkkosivulta linkkiä, jossa on teksti ”Pelit”, mutta sellaista ei löydy.



Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	3	2	1	00:00:14	
All Tests	3	2	1	00:00:14	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
No Tags					

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Robot	3	2	1	00:00:27	
Robot.Browse Products	1	0	1	00:00:17	
Robot.Cart Test	1	1	0	00:00:04	
Robot.Making Orders	1	1	0	00:00:06	

Kuva 7. Robot Framework -testiraportti

8 Verkkokauppatuotteiden tilaaminen

Tässä luvussa käyn läpi, miten Robot Frameworkillä voitaisiin toteuttaa käyttäjätarina ”käyttäjä voi tilata tuotteita” verkkokauppasovellukselle.

8.1 Projektin ja testin pystytys

Lopullinen tiedostorakenne näyttää seuraavalta:

- robot/making_orders.robot
- robot/resources/local.robot
- robot/resources/common_keywords.robot
- robot/resources/common_variables.robot

resources/local.robot

```
1   *** Settings ***
2
3   Library     Selenium2Library  0  implicit_wait=10
4   Resource    common_keywords.robot
5   Resource    common_variables.robot
6
7   *** Variables ***
8
9   ${BROWSER}    firefox
10  ${DOMAIN}     localhost
11  ${PROTOCOL}   http
12  ${PORT}       3000
13  ${SERVER}     ${PROTOCOL}://${DOMAIN}:${PORT}
14
15  *** Keywords ***
16
17  Open Browser And Open Frontpage
18    Open Browser  ${SERVER}  browser=${BROWSER}
19    Maximize Browser Window
20    Location Should Be  ${SERVER}/
```

Esimerkki 6. Ympäristökohtainen asetukset tiedosto

Aloitetaan luomalla projektille asetukset tiedoston nimeltä ”local.robot”. Tiedosto sisältää ympäristöön liittyviä asetuksia. Jos haluttaisiin suorittaa testit tuotantoympäristössä, tällöin voitaisiin luoda tiedoston ”production.robot” ja määrittää sille sopivat asetukset.

Rivillä 3 on lisätty Selenium 2 -lisäosa, joka mahdollistaa kommunikaation selaimen kanssa. Lisäksi riveillä 4 – 5 ladataan kaksi omaa tiedostoa, joissa on kaikkien testien yhteisiä muuttujia ja avainsanoja.

Tiedoston lopusta löytyy avainsana ”Open Browser And Open Frontpage”. Avainsana käyttää hyödyksi edellä määritettyjä muuttujia ja avaa määritetyn verkkosivun Mozilla Firefox -selaimella.

```
making_orders.robot
1   *** Settings ***
2   Resource          resources/${ENVIRONMENT}.robot
3   Suite Setup       Open Browser And Open Frontpage
4   Suite Teardown    Close Browser
```

Esimerkki 7. Suiten asetukset.

Seuraavaksi luodaan tiedoston “making_orders.robot”, joka on tiedosto johon lisätään itse testit. Rivillä 2 ladataan edellisessä esimerkissä määritelty ”resources/local.robot” tiedosto. Tiedosto käyttää globaalia muuttujaa ”ENVIRONMENT”, joka voidaan määrittää testien käynnistys vaiheessa, jotta voidaan valita mitä ympäristöä vasten testit ajetaan. Testit voidaan suorittaa seuraavalla komennolla:

pybot –variable ENVIRONMENT:local .

8.2 Käyttäjätarinan muuttaminen testiksi

```
making_orders.robot
5
6   *** Test Cases ***
7
8   Ordering Items
9     Add Valid Product To Cart
10    Go To Order Page
11    Confirm Subtitle  Order
12    Fill Order Form With Valid Values
13    Send Order
14    Confirm Status Message  Order has been sent!
```

Esimerkki 8. Testitapauksen määrittäminen.

Nyt voidaan kirjoittaa käyttäjätarina “käyttäjä voi tilata tuotteita” testiksi. Rivillä 8 määritetään testin nimeksi ”Ordering Items”. Lisätään testiin avainsana kutsuja, jotka pidetään mielellään selkokielisinä, jotta niitä on helppo ymmärtää. Testin kulku sujuu seuraavasti:

- Lisää tuote ostoskoriin.
- Mene tilaus sivulle.
- Varmista että saavuimme tilaus sivulle.
- Täytä tilauslomake toimivilla tiedoilla.

- Lähetä tilaus.
- Varmista että vastaanotimme onnistuneen tilausviestin.

8.3 Testit valmiiksi

Nyt voidaan suorittaa testit. Testien suoritus antaisi alla olevan virheen, joka kertoo että kyseistä avainsanaa ei löydetty.

No keyword with name 'Add Valid Product To Cart' found.

Hyväksymistestivetoisessa kehityksessä voitaisiin nyt alkaa tekemään ohjelmakoodia tuotteen lisäykselle ostoskoriin sekä toteuttaa tarvittavat RF:n avainsanat.

resources/common_keywords.robot

```
1   *** Keywords ***
2   Select Product  [Arguments]  ${selector}
3     ${product_name} =  Get Text  css=#products p:nth-child(${selector}) a
4     Click Link  css=#products p:nth-child(${selector}) a
5     [return]  ${product_name}
6
7   Add Valid Product To Cart
8     Click '${valid_category}' Category
9     Select Product  1
10    Add Current Product To Cart
11
12  Add Current Product To Cart
13    Click Link  link=Add to cart
```

Esimerkki 9. Avainsanan “Add Valid Product To Cart” toteutus esimerkki

Esimerkissä 9 on mahdollinen toteutus avainsanalle ”Add Valid Product To Cart”. Se käyttää myös kahta muuta avainsanaa: ”Add Current Product To Cart” ja ”Select Product”, jotka on jaettu omiksi avainsanoiksi selkeyden ja uudelleenkäytettävyyden vuoksi.

Huomioitavaa avainsanassa ”Select Product” on, että se palauttaa valitun tuotteen nimen. Palautus tehdään sen vuoksi, että tuotteen nimeä voidaan käyttää vertailuun itse testissä.

Avainsana ”Add Valid Product To Cart” valitsee ennalta määritetyn kategorian, jonka tiedetään aina löytyvän. Sen jälkeen valitaan ensimmäinen tuote ja painetaan sivulla ”Add to cart” nappia.

9 Jatkuva integraatio

Jatkuva integraatio (Continuous integration, CI) on ohjelmistokehityksessä käytetty tapa, jonka tarkoitus on automaattisesti suorittaa sovelluksen testien ajo ja raportoida siitä välittömästi. Päätavoitteena on nopea palautteen saaminen muutoksista, jotta voidaan korjata poikkeamat nopeammin.

CI-järjestelmä toimii yleensä erillisellä tietokoneella sekä mielellään sellaisella, joka on aina päällä. Pääasiassa CI-järjestelmä seuraa versionhallinta järjestelmää ja havaitsee muutokset. Kun muutos tapahtuu, se aloittaa ennalta määritetyn rutiinin, joka yleensä sisältää sovelluksen testauksen.

CI-järjestelmä voidaan laittaa raportoimaan tuloksista, esimerkiksi sähköpostilla kaikille tarvittaville henkilöille. Tämä antaa todella suuren edun, sillä kaikki sähköpostin saavat henkilöt tietävät aina missä tilassa versionhallinnassa oleva ohjelmakoodi on. Lisäksi kun testeissä tapahtuu poikkeama, voidaan se korjata välittömästi. Tällöin poikkeamat ovat myös todennäköisesti pienempiä ja helpommin korjattavia. Jotta CI-järjestelmästä saadaan mahdollisimman paljon etua, se vaatii kehittäjiltä koodin lisäämistä versionhallintaan mahdollisimman usein.

CI-työkaluja voidaan automaattisen testauksen lisäksi käyttää esimerkiksi dokumentaation luontiin tai tuotantojulkaisun suorittamiseen. CI-järjestelmään asioiden automatisointi vähentää kehittäjien työtä sekä varmuutta, sillä ihminen voi unohtaa tehdä tärkeän asian, mutta tietokone suorittaa asian aina samalla tavalla. Esimerkkinä voisi olla tilanne, jossa CI-järjestelmä luo tietokantakaavion automaattisesti joka kerta, kun versionhallintaan tulee muutos. Tällöin tietokantakaavio pysyy aina ajan tasalla, eikä kehittäjien tarvitse murehtia sen ylläpidosta.

9.1 Jatkuva integraatio ja ATDD

ATDD voi olla usein hidasta ja työlästä suorittaa, jolloin ihminen saattaa vahingossa tai tarkoituksella jättää testien suorittamisen. Tällöin testit jäävät jälkeen ja niiden korjaaminen on jatkossa entistä vaikeampaa ja työläämpää. Tämän vuoksi CI-järjestelmä on oiva apuväline hyväksymistestivetoiseen kehitykseen. CI-järjestelmä ei unohda testien ajoa ja se ilmoittaa kehittäjille pieleen menneistä testeistä. Tällöin testit ja itse sovellus voidaan korjata välittömästi, eli ennen kuin ongelma kasvaa.

Käytännössä koko prosessi tapahtuisi seuraavalla tavalla. Kehittäjä lisää uuden ominaisuuden versionhallintajärjestelmään. CI-järjestelmä huomaa muutoksen ja aloittaa sille määritetyn rutiinin suorituksen. Jos rutiinin aikana tapahtuu virhe, rutiinin suoritus lopetetaan ja asiasta ilmoitetaan määritetyllä tavalla kehittäjille. Jos rutiinissa ei tapahdu virheitä, voisi CI-järjestelmä suorittaa automaattisesti esimerkiksi tuotantoversion julkaisun.

9.2 Jenkins

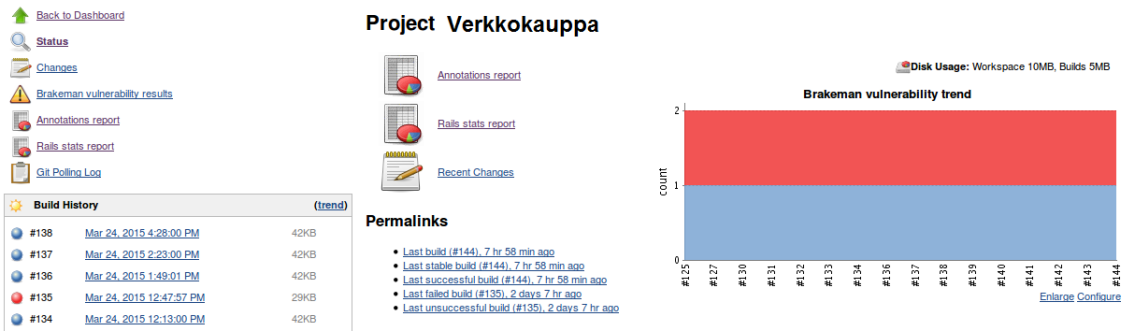
Jenkins on avoimeen lähdekoodiin perustuva CI-työkalu, joka on kirjoitettu Java-kielellä. Verkosta löytyy paljon lisäosia, joilla voidaan lisätä toiminnallisuutta. [14.]

Jenkins rutiinit voidaan käynnistää monilla tavoilla. Rutiinin voi käynnistää esimerkiksi tietty aika, muutos versionhallinta järjestelmässä tai toinen rutiini.

S	W	Name ↓	Last Success	Last Failure	Last Duration	
		Verkkokauppa	3.6 sec - #21	1 min 43 sec - #20	55 ms	
		Verkkokauppa RF	N/A	12 sec - #1	54 ms	

Kuva 8. Jenkinsin projektit näkymä

Kuvasta 8 näemme miten Jenkins listaa eri projektit. Vasemmassa reunassa oleva punainen pallo tarkoittaa, että kyseisen projektin rutiinissa on ollut ongelma ja se on jäänyt kesken.



Kuva 9. Jenkins projektikohtainen näkymä

Kuvassa 6 on Jenkinsin projektikohtainen näkymä. Näkymää voidaan muokata lisäosien avulla. Kuvassa on käytetty Brakeman -lisäosaa, joka näyttää sovelluksen tietoturva haavoittuvuudet kaaviona kuvan oikeassa reunassa. Vasemmalla on osio ”Build History”, josta nähdään kaikki rutiinit, jotka on ajettu, sekä niiden onnistumisen tila.

10 Yhteenveto

Minulle sopi tämä insinöörityö aihe hyvin, sillä halusin oppia lisää ohjelmistotestaamisesta ja tässä työssä sain tutustua kaikkiin sen puoliin yleisellä tasolla sekä hyväksymistestivetoiseen kehitykseen erityisesti. Uskon käyttäväni hyväksymistestivetoista kehitystä tulevaisuudessa projekteissani.

Kävin tutkielmassani yleisesti läpi, mitä sovellustestaus on ja mihin sitä käytetään. Keskityin pääosin hyväksymistestivetoiseen kehitykseen ja selvitin, miksi sitä tulisi hyödyntää sovelluskehitys projekteissa ja mitkä saattavat olla haasteet sen käytössä. Lopuksi tutkin, mitä jatkuva integraatio on ja totesin, että se on melkein välttämätön hyväksymistestivetoisessa kehityksessä, sillä iso ongelma on testivetoisessa kehityksessä on testien ylläpito ja testien suorituksen muistaminen.

Kommunikaatio tuntuu olevan iso ongelma ohjelmistokehitysprojekteissa. Hyväksymistestivetoinen ohjelmistokehitys pakottaa projektin eri osapuolet kommunikoimaan keskenään, mikä on mielestäni todella tärkeä asia ohjelmistoprojektin onnistumisen kannalta.

Hyväksymistestivetoinen ohjelmistokehitys parantaa projektin onnistumisen todennäköisyyttä huomattavasti, mutta kuitenkin siinä on riskinsä. Hyväksymistestivetoisessa ohjelmistokehityksessä on suuret sudenkuopat ja siksi on tärkeää, että projektin jokainen osapuoli on omistautunut noudattamaan sitä täysin.

Olen tyytyväinen tämän insinöörityön lopputulokseen sekä kaikkeen, mitä opin sitä tehdessäni. Sain paremman ymmärryksen ohjelmistotestauksessa käytettävistä eri menetelmistä ja siksi osaan paremmin valita itselle sopivat menetelmät tulevaisuudessa. Lisäksi osaan hyödyntää Robot Frameworkiä hyväksymistestauksessa, jolloin sovelluksistani tulee todennäköisesti luotettavampia ja parempia rakenteeltaan.

Haluan lopuksi kiittää vielä Eficode Oy:tä tämän insinöörityö aiheen antamisesta sekä kaikesta, mitä olen heiltä oppinut, ja tuesta, jota olen heiltä saanut.

Lähteet

1 Why is software testing necessary? Verkkodokumentti. http://istqbexamcertification.com/why-is-testing-necessary/. Luettu 5.1.2015.

2 The Art of Software Testing, 3rd Edition, 2011, Glenford J. Myers, Corey Sandler, Tom Badgett

3 Pollari, Jukka. 2014. Ohjelmistotestaus. Verkkodokumentti. https://publications.theseus.fi/handle/10024/85400/. Luettu 5.1.2015

4 What is unit testing? Verkkodokumentti. http://istqbexamcertification.com/what-is-unit-testing/. Luettu 8.1.2015.

5 Leppäniemi, Jani. 2014. Testaamisen kehittäminen yrityksen ohjelmistokehityksessä. Verkkodokumentti. https://publications.theseus.fi/handle/10024/73071/. Luettu 8.1.2015

6 Acceptance Tests. Verkkodokumentti. http://www.extremeprogramming.org/rules/functionaltests.html/. Luettu 15.12.2014

7 Ketterän ohjelmistokehityksen julistus. Verkkodokumentti. http://agilemanifesto.org/iso/fi/. Luettu 28.3.2015

8 Continuous integration. Verkkodokumentti. http://searchsoftwarequality.techtarget.com/definition/continuous-integration/. Luettu 10.3.2015

9 Nieminen, Arttu. 2014. Käyttäytymislähtöisen kehityksen työkalut. Verkkodokumentti. https://www.theseus.fi/handle/10024/72908/. Luettu 10.3.2015

10 Kärkinen, Simo. 2013. Käyttäytymislähtöinen ohjelmistokehitys. Verkkodokumentti. https://www.theseus.fi/handle/10024/58421/. Luettu 28.2.2015

11 Koudelia, Nikolai. 2011. Acceptance Test-Driven Development. Verkkodokumentti. https://jyx.jyu.fi/dspace/bitstream/handle/123456789/37392/URN%3aNBN%3afi%3ajyu-201202161200.pdf?sequence=1/. Luettu 14.1.2015

12 Acceptance Test-Driven Development with Robot Framework. Verkkodokumentti. http://wiki.robotframework.googlecode.com/hg/publications/ATDD_with_RobotFramework.pdf/. Luettu 8.1.2015.

13 Robot Framework. Verkkodokumentti. http://robotframework.org/. Luettu 10.3.2015.

14 Jenkins. Verkkodokumentti. http://searchsoftwarequality.techtarget.com/definition/Jenkins/. Luettu 10.3.2015.

15 Running Agile in a small team of 2 to 3. Verkkodokumentti. http://www.dancourse.co.uk/2014/08/running-agile-in-a-small-team-of-2-to-3/. Luettu 8.1.2015.

16 White-Box Unit testing-A Bottom-Up Approach of Software Testing. Verkkodokumentti. http://www.softwaretestinggenius.com/white-box-unit-testing-a-bottom-up-approach-of-software-testing. Luettu 16.2.2015.

17 Test-driven development. Verkkodokumentti. http://en.wikipedia.org/wiki/Test-driven_development. Luettu 16.2.2015

18 Agile Acceptance Test Driven Development. Verkkodokumentti. http://blog.simplilearn.com/project-management/agile-acceptance-test-driven-development. Luettu 16.2.2015.

19 Integration Testing. Verkkodokumentti. http://www.techopedia.com/definition/7751/integration-testing. Luettu 17.2.2015.

20 TDD. Verkkodokumentti. http://guide.agilealliance.org/guide/tdd.html. Luettu 17.2.2015.

Robot Framework verkkokauppa esimerkki

RF ostoskori testi

cart_test.robot

```
*** Settings ***
Resource          resources/${ENVIRONMENT}.robot
Suite Setup       Open Browser And Open Frontpage
Suite Teardown    Close Browser

*** Test Cases ***

User Can Add Products To Cart
  Click '${valid_category}' Category
  ${product_name} =  Select Product  1
  Add Current Product To Cart
  Go To Cart
  Cart Contains Product  ${product_name}

*** Keywords ***
Cart Contains Product  [Arguments]  ${product_name}
  Table Should Contain  css=table#products  ${product_name}
```

RF tuotteiden selaus testi

browse_products.robot

```
*** Settings ***
Resource          resources/${ENVIRONMENT}.robot
Suite Setup       Open Browser And Open Frontpage
Suite Teardown    Close Browser

*** Test Cases ***
Browse Categories
  Click 'Cameras' Category
  Confirm 'Cameras' Page Subtitle
  Click 'Games' Category
  Confirm 'Games' Page Subtitle

*** Keywords ***
Confirm '${category}' Page Subtitle
  Wait Until Keyword Succeeds  5 s  .5 s  Confirm Subtitle  ${category}
```

RF tuotteiden tilaus testi

making_orders.robot

```
*** Settings ***
Resource          resources/${ENVIRONMENT}.robot
Suite Setup       Open Browser And Open Frontpage
Suite Teardown    Close Browser

*** Test Cases ***

Ordering Items
  Add Valid Product To Cart
  Go To Order Page
  Confirm Subtitle  Order
  Fill Order Form With Valid Values
  Send Order
  Confirm Status Message  Order has been sent!

*** Keywords ***
Go To Order Page
  Go To Cart
  Click Link  link=Order

Fill Order Form With Valid Values
  Fill Name Field  Maija Meikäläinen
  Fill Street Address Field  Testikatu 12
  Fill City Field  Helsinki
  Fill Postcode Field  00100
  Fill Telephone Field  050-1234567

Send Order
  Click Button  name=commit

Fill Name Field  [Arguments]  ${name}
  Input Text  id=order_name  ${name}

Fill Street Address Field  [Arguments]  ${street_address}
  Input Text  id=order_street_address  ${street_address}

Fill City Field  [Arguments]  ${city}
  Input Text  id=order_city  ${city}

Fill Postcode Field  [Arguments]  ${postcode}
  Input Text  id=order_postcode  ${postcode}

Fill Telephone Field  [Arguments]  ${Telephone}
  Input Text  id=order_telephone  ${Telephone}
```

RF ympäristö asetukset

resources/local.robot

```
1   *** Settings ***
2
3   Library     Selenium2Library  0  implicit_wait=10
4   Resource    common_keywords.robot
5   Resource    common_variables.robot
6
7   *** Variables ***
8
9   ${BROWSER}    firefox
10  ${DOMAIN}     localhost
11  ${PROTOCOL}   http
12  ${PORT}       3000
13  ${SERVER}     ${PROTOCOL}://${DOMAIN}:${PORT}
14
15  *** Keywords ***
16
17  Open Browser And Open Frontpage
18    Open Browser  ${SERVER}  browser=${BROWSER}
19    Maximize Browser Window
20    Location Should Be  ${SERVER}/
```

RF yhteiset avainsanat

resources/common_keywords.robot

```
*** Keywords ***
Select Product  [Arguments]  ${selector}
  ${product_name} =  Get Text  css=#products p:nth-child(${selector}) a
  Click Link  css=#products p:nth-child(${selector}) a
  [return]  ${product_name}

Add Valid Product To Cart
  Click '${valid_category}' Category
  Select Product  1
  Add Current Product To Cart

Add Current Product To Cart
  Click Link  link=Add to cart

Go To Cart
  Click Link  css=#cart a:first-child

Confirm Subtitle  [Arguments]  ${text}
  ${subtitle} =  Get Text  id=subtitle
  Should End With  ${text}  ${subtitle}

Confirm Status Message  [Arguments]  ${text}
  Element Should Contain  css=#flash div  ${text}

Click '${category}' Category
  Click Link  css=#categories a.${category}
```

RF yhteiset muuttujat

resources/common_variables.robot

```
*** Variables ***
${valid_gategory}  Cameras
```